LZ based compression benchmark on PE files

Zsombor Paróczi

Abstract: The key element in runtime compression is the compression algorithm, that is used during processing. It has to be small in enough in decompression bytecode size to fit in the final executable, yet have to provide the best compression ratio. In our work we benchmark the top LZ based compression methods on Windows PE files (both exe and dll files), and present the results including the decompression overhead and the compression rates.

Keywords: lz based compression, compression benchmark, PE benchmark

Introduction

During runtime executable compression an already compiled executable is modified in ways, that it still retains the ability to execute, yet the transformation produces smaller file size. The transformations usually exists from multiple steps, changing the structure of the executable by removing unused bytes, adding a compression layer or modifying the code in itself. During the code modifications the actual bytecode can change, or remain the same depending on the modification itself.

In the world of x86 (or even x86-64) PE compression there are only a few benchmarks, since the ever growing storage capacity makes this field less important. Yet in new fields, like IOT and wearable electronics every application uses some kind of compression, Android apk-s are always compressed by a simple gzip compression. There are two mayor benchmarks for PE compression available today, the Maximum Compression benchmark collection [1] includes two PE files, one DLL and one EXE, and the Pe Compression Test [2] has four exe files. We will use the exe files during our benchmark, referred as small corpus. More detailed results we have a self-collected corpus of 200 PE files, referred as large corpus.

When approaching a new way to create executable compression, one should consider three main factors. The first is the actual compression rate of the algorithms, since it will have the biggest effect on larger files. The second is the overhead in terms of extra bytecode within the executable, since the decompression algorithm have to be included in the newly generated file, using large pre-generated dictionary is not an option. This is especially important for small (< 100kb) executables. The third one has the lowest priority, but still important: the decompression speed. The decompression method should not require a lot of time to run, even on a resource limited machine. This eliminates whole families of compressions, like neural network based (PAQ family) compressions.

Split-stream methods are well know in the executable compression world, these algorithms take advantage of the structural information of the bytecode itself, separating the opcode from all the modification flags. We used a reference implementation from the packer, kkrunchy [5].

LZ based compression methods

Compression method	Version	Source
aPlib	1.1.1	http://ibsensoftware.com/products_aPLib.html
Lzlib	1.10	https://www.nongnu.org/lzip/lzlib.html
LZMA	9.35	https://www.7-zip.org/sdk.html
Zopfli	2017-07-07	https://github.com/google/zopfli
Zstandard	1.3.3	https://facebook.github.io/zstd/
CSC	2016-10-13	https://github.com/fusiyuan2010/CSC
Brotli	1.0.3	https://github.com/google/brotli

Table 1: Libraries used during the benchmark

LZ based compression methods (LZ77/LZSS/LZMA families) are well fitted for this compression task, since they usually have relatively small memory requirement (< 64 Mb), they use Lempel-Ziv compression methods [3] and maybe some Huffman tables or hidden Markov model based approaches. These methods also result in simple algorithms, resulting in small size in terms of decompression byte-code. During the last few years there are a lot of new LZ based compression methods, the mayor ones are Zstandard (zstd) from Facebook and Zopfli from Google. The selected libraries can be seen on Table 1., these are the top LZ familly libraries for generic purpose compression regarding to an extensive LZ benchmark [4].

Benchmark

During the benchmark we constructed a system, which is capable of extracting different sections from the executables, apply split-stream and a compression on it to create a well detailed benchmark result. During the benchmark we run each compression method on each section, then run each compression method with split-stream on executable sections.

Results: compression ratio

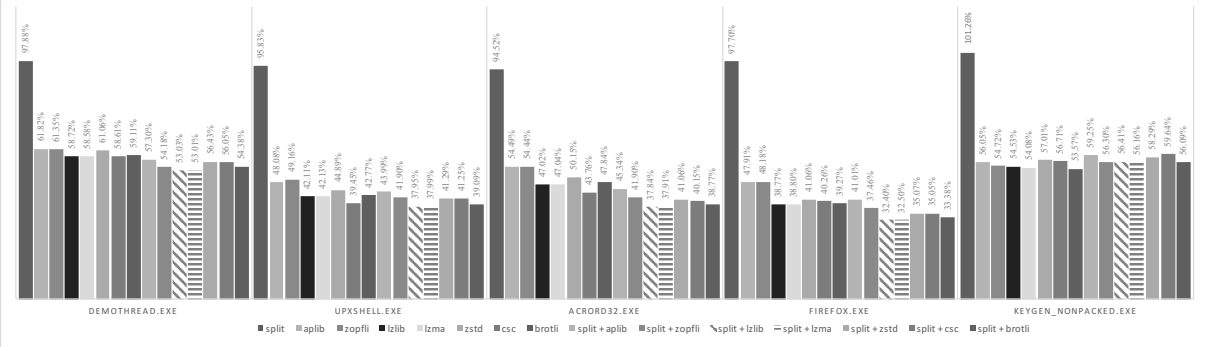

Figure 1: Resulting section size compered to the original on the small corpus files

The detailed results for each test case on the small corpus can be seen on Figure 1. As you can see applying split-stream before the compression is useful in most of the cases (except for the smallest executable, which suffered from the overhead of this method - splitting 1 byte instructions into base instruction + mod flags). The rates for each compression varies between test cases, but Lzlib, LZMA, Brotli are clearly the best for the small corpus, followed by zstd, CSC, Zopfli and aPlib. There is a constant improvement when using split-stream. Only for really small executable aPlib is the best, due to the simplicity of the algorithm itself. All of these results were verified during our large corpus benchmark.

aplib	lzma	s + aplib	s + zopfli	s + lzlib	s + lzma	s + zstd	s + csc	s + brotli
47%	42%	44%	41%	40%	39%	42%	42%	40%

Table 2: Average compression rates on code section

aplib	zopfli	lzlib	lzma	zstd	csc	brotli
40%	37%	35%	34%	38%	35%	33%

Table 3: Average compression rates on non-code section

The actual compression rates on the large corpus can be seen on Table 2. and 3. (split-stream is annotated as s). As you can see the ratio between each compression rate on average is really small, for code sections split-stream really helps. For code section LZMA, Lzlib and Brotli are the best, followed by Zopfli and CSC. For non-code section we had a larger variety of results, since the non-code sections can contain any datatype. Since it has a more loose structure and less density, the compression rates are higher. It is interesting, that Brotli was the winner in these tests, but as it turned out Brotli has a large dictionary prebuilt into the algorithm, that helps with compressing text. LZMA, LZlib, CSC produced just 1-2% lower rates, followed by zstd and Zopfli. Obviously aPlib was the worst in both tests, since it contains the most simple algorithm for compression. Since the PE sections tend to be less then 3 Mb, the larger the section the more compression rate we can achieve but to the lower entropy.

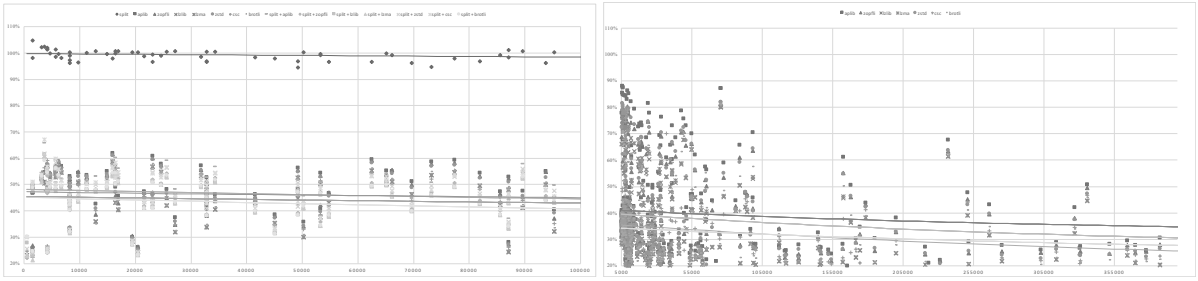

Figure 2: Compression rates vs file size: left the code section, right the non-code section results

Results: final file size with decompression bytecode

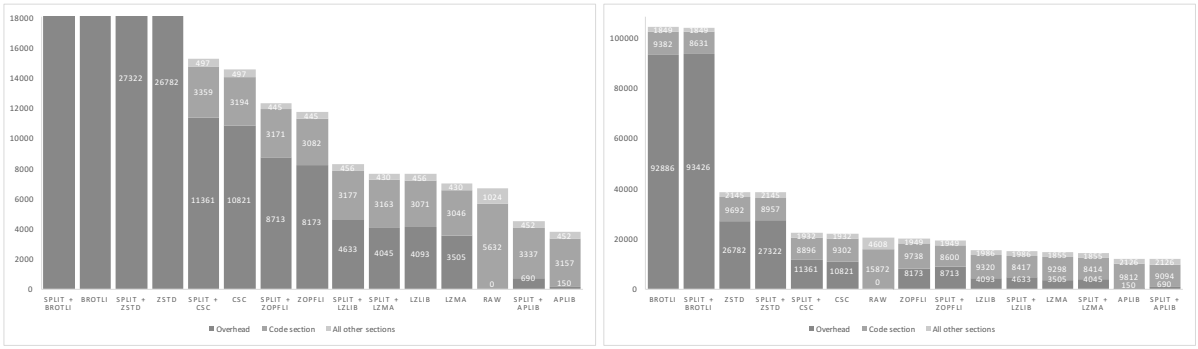

Figure 3: Final executable size: keygen_unpacked.exe, DemoThread.exe

Since the decompression code has to be included in the final executable, we also benchmarked how the decompression overhead code effects the final file size. As you can see on Figure 3. for smaller executables the overhead is what really defines the final result. All of the decompression methods were packed with aPlib, since aPlib has a decompression code size of 150 bytes, and above 1.000 bytes it is better to compress the decompression code with aPlib. Some of the more complex methods (namely zstd, Brotli, CSC) has relatively large data tables in the decompression code. Same goes for the split-stream code, which is above 1kByte uncompressed, and 540 byte compressed with aPlib.

Our final results suggest, that there is no "golden" LZ based compression with split-stream method for all the executables. We consider 3 categories based on the executable size: for small files (< 50kB) size aPlib is the clear winner with 150 byte decompression code, maybe with split-stream if the executable section is large. For medium size (< 500 kB) split-stream with aPlib or split-stream with LZMA (aPlib compressed) should be used. For larger files split-stream with LZMA (aPlib compressed) or split-stream with Lzlib (aPlib compressed) should be used. You can see the the best performing algorithm on Figure 5. for the large corpus. For some special cases each combination can be the winner in regards of final compression size. CSC (without split-stream), Lzlib (without split-stream) and LZMA (without split-stream) can outperform the others in some cases.

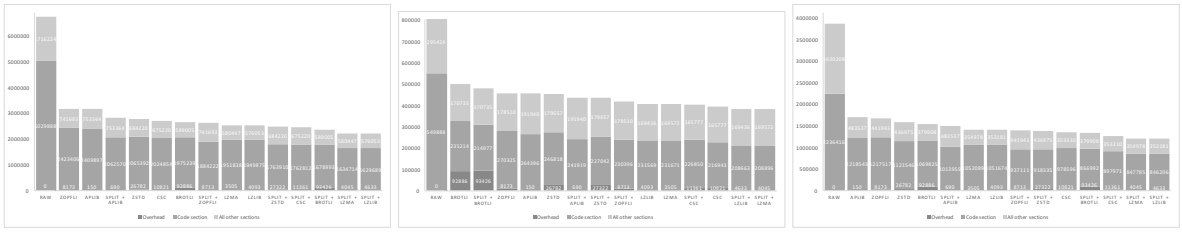

Figure 4: Final executable size: Firefox.exe, UPXShell.exe, acrord32.exe

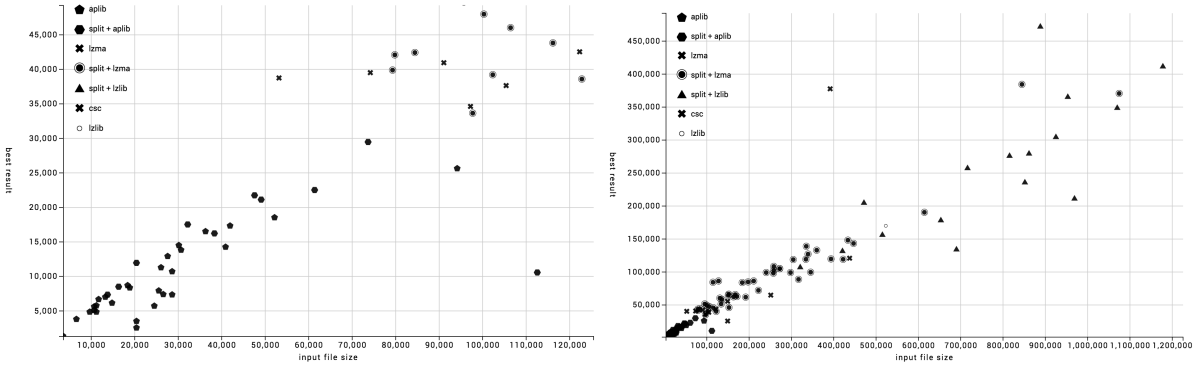

Figure 5: Raw and compressed file size using the best method

References

[1] Lossless data compression software benchmarks / comparisons, *https://www.maximumcompression.com/* (Visited 2018 2018-03-04).

[2] Pe Compression test by Ernani Weber *http://pect.atspace.com/* (Visited 2018-03-04).

[3] Ziv J., Lempel A., A universal algorithm for sequential data compression, *IEEE Trans. on Inf. Th. IT-23* (1977) 337-343.

[4] LZbench *https://github.com/inikep/lzbench/* (Visited 2018-03-04).

[5] Fabian Giesen. Working with compression. *Breakpoint conference* (2006)